

Understanding ActiveX and COM

The Component Object Model (COM) has caused more confusion since its inception than any other programming technology that has ever appeared on the horizon. Many programmers feel like blind men trying to describe an elephant and most of them don't feel they do any better in figuring out what COM really is. The fact that Microsoft has made COM a moving object and keeps changing the definition of what COM is, based on marketing needs, doesn't help matters much. For example, Microsoft tried to change all COM technologies into some form of ActiveX as little as three years ago, yet ActiveX is now described as simply an application component technology—a part of a larger whole. As a result, there's probably less work done with this very useful technology than could be; and yet the results that programmers do get are nothing less than amazing. In other words, COM is worth the effort to learn, but you probably won't learn everything about it in a few days.

This chapter is designed to give you a simple overview of COM from a theoretical perspective. It answers the question like what COM is and what it can do for you within an application. We'll also talk about the various forms of COM and what they mean to you as a programmer. For example, we'll look at how DCOM differs from standard COM. We'll also discuss two very important COM features; encapsulation and code reusability. You'll learn how to work with interfaces and how to use them to create new objects. Finally, we'll look at one of the most important diagnostic and research aids at your disposal—OLE/COM Object Viewer.

11 CHAPTER



In This Chapter

Discovering the benefits of using COM versus other technologies

Understanding how the various COM classes come into play

Learning about COM, ActiveX, OLE, DCOM, and COM+

Finding free components that you never knew existed

Understanding the benefits of encapsulation and code reuse

Creating objects using interfaces

Working with COM using the OLE/COM Object Viewer utility



**Note**

Although this chapter will provide you with the information you need to work with the examples in the next chapter, it's important to realize that a single chapter can't replace what entire books hope to cover but can't. There are a lot of questions that this book won't answer. If you don't have at least a passing familiarity with COM, you'll need to spend some time learning some of the basics before this chapter will make much sense. We also won't spend any time covering the really advanced topics that are more issues of debate than of practical importance. In other words, this chapter will cover issues of practical value that you'll need to know about as the book progresses.

COM Benefits

In the beginning, there was spaghetti code, and though the spaghetti code worked, no one could understand it. Then came an age of enlightenment when programmers learned about the usefulness of modular programming techniques. Although coding was still difficult, it wasn't impossible, and someone other than the programmer could actually understand the code. Modular programming first gave programmers the opportunity to reuse code. If the programmer could write a function or procedure in a generic manner, then the function or procedure could appear in another application, easing the programmer's burden.

Next came libraries, which allowed the programmer to place compiled and debugged code in a separate file to reduce the complexity of the application, making it even easier to understand. Linking a library into an application to perform generic tasks meant that the programmer didn't need to worry about the details of the function or procedure, only the calling syntax that was required. Unfortunately, the library had to get added to the application during the link cycle, which meant that updates to a library file also meant updating a lot of applications.

Finally, the dynamic link library (DLL) arrived on the scene, which freed the programmer from having to incorporate the code into the application. A DLL improves application development in a number of ways. For one thing, updating a DLL doesn't necessarily mean any update to the application since the DLL gets loaded separately from the application. In addition, DLLs use memory more efficiently because more than one application can access the code they contain—you need only one copy of the DLL in memory instead of the multiple copies that libraries require.

Now we have COM, which incorporates every advance in technology that has come before and makes it even easier to write bulletproof applications. COM provides the programmer with a lot of functionality. It incorporates the notion of code reuse and reduced memory consumption. Updating a COM application doesn't necessarily entail updating the applications that rely on the COM application as well. However, COM provides more than these simple benefits—things that we now take for granted. The following sections outline the other benefits that COM provides.

Language Independence

Depending on the programming language you use, a string might consist of a null terminated array of characters or an array of characters in which the first array element contains the number of characters that follow. This is just one example of how languages can differ. In the past, you had to use libraries that were developed for your programming language or convert your data and calling methodology to meet the library's expectations. COM allows you to create libraries of routines that don't rely on the language used to develop them.

Platform Independence

Theoretically, COM allows you to move your application source code from platform to platform without any major recoding. Because the interface for COM is always the same, an application can use the same calls on every platform. COM automatically takes care of the hardware differences between platforms, effectively isolating the programmer from the hardware and reducing application complexity. Unfortunately, this is one area where much of the potential for COM has not been yet realized.

Extensibility

COM is an object-oriented programming (OOP) methodology, which means that it allows you to create a new object based on an existing object type. In other words, if there's a generic bit of code that does almost what you want, but not quite, you can create a new version of the code that incorporates all of the features you need, without rewriting everything from scratch. In addition, the original object can appear in a library, while the new object appears in your code.

Encapsulation

There are actually two benefits to encapsulation. The first is protection. When using COM, you get access to the properties and methods that the original programmer wants to make public, and no more. This means that the objects you create are less likely to fail or produce unexpected results — at least not if you use the object with the vendor's guidelines in mind (we'll see later that nothing is perfect).

The second encapsulation benefit deals with the idea of an interface. The interface defines how the client (the application using the component) and the server (the component itself) interact. As long as the interface doesn't change, then the client and server will interact in the same way. This means you can change the internals of a component to fix bugs or optimize performance without having to recompile the client as long as you don't change the interface. The reverse is also true.

Locality

Older applications relied on having their resources local to the executing machine. In addition, all of those resources had to reside in the same process as the application itself. COM allows a component to execute within the same process as the application it services, in a different process on the same machine, or even on a remote machine (given certain circumstances that we'll discuss later in this chapter).

Classes of COM

COM didn't happen overnight, it evolved over a period of years. Unfortunately, the older versions of COM didn't go away immediately, they've stayed around in the form of older applications and in new applications that don't require the full capabilities that COM provides. What this means to you, as a programmer, is that you'll actually deal with several classes of COM. Every time Microsoft comes up with a new set of features, they add them to the existing version of COM and come up with an entirely different name. In essence, each COM class builds on the capabilities of its predecessor.

The following sections will help you learn about the different classes of COM that you may run into as you develop applications. What you'll see is a history of how COM developed and get a better understanding of why some things are the way they are. In some cases, you might gain a new understanding of how some oddity got introduced into what appears to be an otherwise logical method of promoting code reuse and application modularity.

OLE

There are actually two versions of Object Linking and Embedding (OLE) used within the various versions of Windows. The first version of OLE relies on dynamic data exchange (DDE), whereas the second version relies on a very early version of COM. We'll see why this distinction is important as the section progresses.

The Windows registry maintains the entries for the two versions of OLE in completely separate sections. The OLE1 entries appear in the file association section of the HKEY_CLASSES_ROOT hive. One of the entries in the OLE1 section of the registry does refer to the OLE2 entry, thus linking them together. Figure 11-1 shows a typical OLE1 registry entry. Notice that this entry relies on DDE scripting to accomplish its task.



Figure 11-1: OLE1 registry entries rely on DDE.

The OLE2 entries appear under the `HKEY_CLASSES_ROOT\CLSID` key as part of the class identifier entries (those 128-bit numbers that we've been using throughout the book). Figure 11-2 shows a typical OLE2 registry entry. Notice that this entry relies on COM. In other words, OLE2 and COM are intrinsically linked.

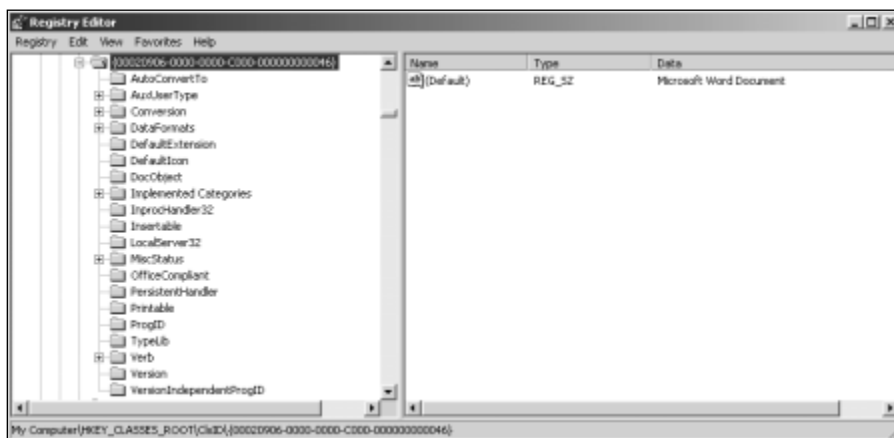


Figure 11-2: OLE2 registry entries rely on COM.

The following sections discuss the differences between OLE1 and OLE2, and why they're important to you as a programmer. In addition, you'll see how OLE2 evolved naturally from OLE1 as part of Microsoft's learning process. Finally, you'll see why OLE2 was a significant start for Microsoft and how it led to the COM technologies that we use today.

OLE1 and DDE

Let's talk about DDE for a moment. DDE used to be the way to transfer data between applications and make them work together (sort of), then OLE arrived on the scene and made life easier, but COM (or one of its predecessors) is the way to go today. There are some significant problems with using DDE. The most significant of these is that it creates a static link, much like the one you get using cut-and-paste. The fact that it provides a stable macro language that you can use to open files and perform other fancy maneuvers from the command line doesn't change much.

DDE is a messaging protocol. It sends a message from one application to another and asks it to do something. Originally, DDE was supposed to provide the means to open another application and copy some data to the Clipboard. It also did other chores such as printing files.

A DDE macro contains part DDE and part application macro language. This is another problem with using it. Not only do you need to learn the native language of the application you're using and DDE itself, but you also have to learn the macro language for the server application. Needless to say, DDE didn't get the kind of reception that Microsoft had originally hoped for. DDE is simply too hard for the average user to consider using. In fact, even some programmers find it difficult to use (unless they use it on a regular basis).

OLE2 and COM

As previously stated, OLE2 doesn't rely on DDE with its static links and difficult macro language; it relies on COM. Using COM allows OLE2 to deal with documents as objects and makes it possible for client and server to improve independently of each other by maintaining the same interface. To give you a better idea of how this works, look at the DocFile Viewer display shown in Figure 11-3. Notice that this document contains three objects, two of which are sound files and one of which is a Microsoft Word document. As you can see, the effect of COM is real when it comes to working with OLE2.

It's because of the COM difference that OLE2 is now used for most application links rather than OLE1. You'll still find both DDE and OLE1 in Windows in some form, but not as a major component. The following sections give you an idea of all of the ways that using COM for OLE2 improved the situation.

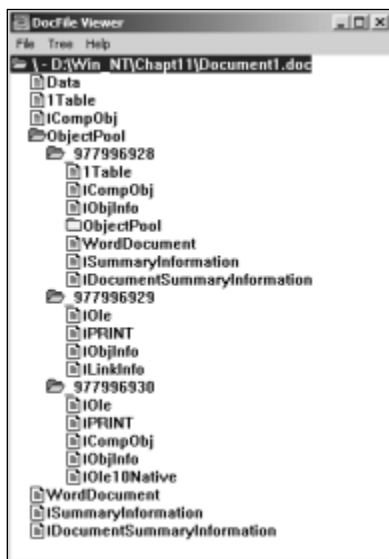


Figure 11-3: COM is a very real part of OLE2, as shown by the objects in this document file.

Visual editing

One of the problems with OLE1 was that the user's train of thought got disrupted every time he or she needed to make a change to an object. The reason is simple: OLE1 was forced to use the DDE macro language to load the original application and display the object in the originating application's window for editing. OLE2 allows visual (in-place) editing because it uses COM, which supports both in-process and out-of-process servers. Instead of opening a new window, the server application merely overlays its toolbar, menu structure, and controls, with those of the client. The user simply sees a change in tools, not a change in applications. As a result, the transition between documents is less noticeable.

Nested objects

OLE1 allowed you to place one object at a time in the container document. An object couldn't become a container; all the objects existed as a single layer within the container. OLE2 treats every potential container as just that — a container. It doesn't matter how many containers you place inside a container or how many ways you stack them. To get a better idea of how nesting will help you, look at the way Windows versions starting with Windows 95 implements folders. You can treat OLE2 container objects the same way.

Drag and drop

You used to cut or copy an object in the server application and then place it in the client by using the Paste Special command. This option still works. However, OLE2 provides a new method of creating links to other documents. You can simply grab the object and move it wherever you want. It becomes linked wherever you decide to drop it.

Storage-independent links

OLE2 enables you to create links to other documents, even if they aren't physically located on the local drive. It implements the links using an LRPC (light remote procedure call) mechanism. Unfortunately, this linking mechanism has limitations. For example, you'll find that it works fine with some peer-to-peer networks, but works only marginally well with other network types. The next revision of OLE is supposed to fix this problem by supporting RPCs (remote procedure calls).

Adaptable links

If you moved any of the files required to create a compound document under OLE1, all the links were destroyed, and you had to re-create them. OLE1 stored the full path, including drive, to the linked data. OLE2 stores only enough path information to maintain the link. If you create links between two files in the same directory, you can move these two files anywhere on the drive, and OLE2 can maintain the link. The only criterion for maintaining a link under OLE2 is that the relative path remains the same.

OLE automation

This particular feature is tied into Visual Basic for Applications (VBA), which is a subset of the Visual Basic programming language. VBA defines a standard interface for talking with the server application. This allows the client application to send commands to the server that will change the contents of an object indirectly. OLE automation is the direct descendent of the DDE macro language that many applications still use (with a COM twist of course). The big difference from the user's perspective is that DDE macros were difficult to write and very prone to error. VBA is the native language of the application and is consistent across platforms.

Version management

Have you ever received a document from someone only to find that part of it wouldn't work with your software? OLE2 can store the application name and version number as part of the link (they're stored as object properties). If an application developer implements this feature correctly, a server (or client, for that matter) will detect an old version of a file and ask whether you want to update it. This means that you'll never have an old file sitting around just waiting to make life difficult.

Object conversion

Your friend uses Excel and you use Lotus 1-2-3, yet you need to share OLE documents containing spreadsheets. One of you could go through the inconvenience and expense of changing to the other person's application and document format, or you can use OLE2, which can probably solve this problem without such a change. Object conversion allows Excel to act as a server for a compound document containing a Lotus 1-2-3 object. All you need to do is select the Convert option from the object menu. (Obviously, the application must support the data format before the Convert option will become available.)

Optimized object storage

This feature allows the linked documents to stay on disk until needed. That way, Windows doesn't need to load every application and data file required to support the compound document. In most cases, Windows uses a buffer-zone technique. A word processor might keep the applications and objects required for the preceding, current, and next page in memory. The rest of the objects stay on disk, greatly reducing the memory footprint of a compound document in some cases.

OLE Support File Requirements

There's a whole group of files in either the \SYSTEM or SYSTEM32 folder that provide support for OLE2. From a programmer's perspective, you seldom need to know about these files unless you either suffer a corrupt file or need to check on the version of OLE that you're using. The following list provides some details on the tasks each file performs. You can use this list if you ever run into a problem with corruption or if you would simply like to know what level of support you can expect from a certain application. The presence or absence of these files might indicate problems with your installation as well.

- ♦ OLE2.DLL: This dynamic link library (DLL) provides some "foundation" functions. It's the first DLL that gets called when any type of OLE2 call is made by an application. (This is the 16-bit version of the file—we'll discuss 32-bit version files further down in this list.)
- ♦ OLECLI.DLL: This file contains all of the basic client code your application needs to implement OLE2 client calls. Your application uses this file as a base for building its own client features.
- ♦ OLESRV.DLL: This file contains all of the basic server code your application needs to implement OLE2 server calls. Like the client code, this DLL won't provide everything. Your application uses it as a basis for building its own set of features.
- ♦ OLE2CONV.DLL: This file provides the generic routines a program needs to convert an object to the client program's native format.
- ♦ OLE2DISP.DLL: Every OLE client application uses this program to help it display the objects it contains.

Continued

(continued)

- ♦ OLE2NLS.DLL: Most versions of Windows provide National Language Support (NLS). This program helps OLE keep pace with the rest of Windows in providing support for other languages.
- ♦ OLE2.REG: You can import this Registry file into your Registry to install OLE2 support. In most cases, your application will do this automatically, so you don't need to worry about it. The only time you'll need to use it is when you can't get OLE2 to work and you discover that the Registry doesn't contain the correct entries.
- ♦ MCIIOLE.DLL: Sounds and other multimedia require special handling under Windows. This special DLL provides the support that an application needs to handle a sound or other multimedia objects.
- ♦ OLE32.DLL: A whole group of OLE files in the \SYSTEM or SYSTEM32 directory have a "32" somewhere in their names. These files provide the same services as their 16-bit counterparts to 32-bit applications. OLE32.DLL is the main 32-bit OLE2 DLL. You'll also find other 32-bit files that contain 32-bit support for all of the other OLE features in this list.
- ♦ MFCOLEUI.DLL: This file (and any with similar names) provides the Microsoft Foundation Classes (MFC) support interface to OLE. Normally, this file is used only by C++ programmers.

COM

In the previous section, we talked about OLE2's reliance upon COM. We looked at how using COM as a basis rather than DDE made OLE2 a much better product than OLE1. OLE2 is actually a specific implementation of COM; it's not the whole picture. COM is also used as the basis for controls, compound documents, automation, as well as data transfer, storage, and naming.

It's important to understand that COM is actually a specification—a documented method for dealing with the interaction between clients and servers. Some people use the term *programming model* to refer to what COM represents. In other words, COM tells you what needs to happen, not necessarily how to do it. COM represents a programming methodology, not an actual programming implementation. As such, many details are left to the implementer rather than defined as part of the specification itself.



Microsoft has introduced a plethora of new OLE-related technologies such as ActiveX, DCOM, and COM+ over the past few years. Trying to keep up with these technologies could prove daunting for even the best programmer. Not only is Microsoft constantly evolving COM, but there are always new technologies and developments within those technologies to learn about. The general page for all of Microsoft's current COM technologies is at <http://www.microsoft.com/com/default.asp>.

COM is a Microsoft-specific extension of object-oriented programming (OOP) techniques. As such, the central idea of COM is the object, which is a programming method that attempts to model applications after the real world. Objects have the following components (at least when referred to with respect to COM):

- ♦ **Methods:** Things you can do with the object. A method enables you to perform a specific task with the object. It may rely on the values of any properties that the object exposes and may require you to provide values as input.
- ♦ **Properties:** Specifications that determine what the object looks like and how it interacts with the user. A property is a value that affects the internal operation of the object and determines what it will do when that application calls any associated methods.
- ♦ **Events:** An object-specific occurrence that alerts the application to respond to a user or internal action. Examples of user actions include mouse clicks and key presses. Internal actions cover a lot of territory, but are normally associated with some type of status or error-reporting need.
- ♦ **Interfaces:** A convenient grouping of methods and properties. Gaining access to an interface allows you to use the methods and properties that it contains.
- ♦ **Connection Point:** The physical link between the server and the client that allows the object to signal an event.

Overall, the purpose of COM is to hide the underlying complexity of a programming task and allow the user of the object to deal with the task at a higher level of abstraction. For example, a programmer may not care how a string is created or manipulated internally, but cares about how the string appears externally. The fact that a string object allows you to make all of the characters uppercase is important; how the object performs this transition isn't. In other words, COM is a results-oriented programming methodology in which the client doesn't care how the server performs a task, and the server doesn't care what the client does with the results.

Objects are also easier to work with than a library because an object is a cohesive whole that includes both code and data in a single package. When you create a COM object, you don't need to specify the internal arrangement of that object, create variables that get used internally within the object, or worry about things like the programming language used by the object designer. All you need to know is that the object has a certain method and that calling the method produces a certain result given certain property values and method arguments as input.

Microsoft set specific goals for COM. These goals enable COM to provide the level of services that Windows requires without exacting such a high-performance penalty that the operating system runs too slowly. The following list tells you about these goals.

Binary Compatibility	Every COM object uses the same calling convention, which means that any client can call any object as long as they both use a standard COM interface. COM relies on double indirection. The client holds a pointer to a virtual table of object functions, which, in turn, point to the actual functions within the component object. Using a virtual table allows the server to interact with multiple clients at the same time. Even components use this methodology. One component will never call another component directly; the interface is used for all communication.
Strong Typing	If the return value of a function is ambiguous, then the client may not interpret the result correctly. Likewise, if the client provides input values of unknown type, the server may not be able to function properly. Strong typing ensures that clients and servers are both using the same basis to pass information.
Base Interface	A client may not know what services that a server can provide. The use of a base interface allows the client to query the server and ask it what services it provides in the way of properties and methods. In addition, a query can determine the name of the server-specific interface used to bundle these methods and properties. All COM components must support the IUnknown interface, and all clients call that interface to learn more about the component. We'll discuss the IUnknown interface in the "Working with Interfaces" section of this chapter.
Reference Counting	One or more clients could create more than one instance of a particular server. Reference counting enables the server object to keep track of how many instances of itself exist. When there aren't any more active instances, the server object can delete itself.
Unique Identification	Objects can exist on more than one machine, and object confusion would lead to unpredictable results from client requests. The globally unique identifier (GUID) allows COM to uniquely identify every object using a unique 128-bit number. GUIDs actually conform to an Open Software Foundation standard called universally unique identifiers (UUID). This standard ensures that every GUID is unique, no matter where it's generated.
Component Loading	Every component uses the same loader to load itself into memory. This component loader also tracks interactions with objects on other machines.

As you can see, COM represents a fundamental shift from procedural languages. When COM is combined with the Windows event-driven programming methodology, the programmer is left with an environment that's easy to use. The user, in turn, is left with a responsive environment that should be able to react almost immediately to most input.

ActiveX

ActiveX is a COM derivative. In other words, like OLE2, ActiveX relies on COM as a defining standard. There's been a lot of confusion about ActiveX controls in the past, and part of the problem stems from Microsoft's continuing redefinition of all terms related to COM. The following sections are designed to help you gain a fuller appreciation of precisely what ActiveX is and how it relates to COM as a whole. We'll define what an ActiveX control is, and, more importantly, what it isn't (at least in terms of how ActiveX is relevant in this book).



You don't have to learn about ActiveX by yourself. If you'd like some help from fellow programmers, check out the `microsoft.public.activex` newsgroup folder. This folder contains a wealth of newsgroups you can join to get various kinds of COM help, though they do specialize in ActiveX support. A great place to learn about the use of ActiveX on the Internet using current browser technology is the `microsoft.public.inetexplorer.ie4.activex_ctrl` newsgroup. Many component newsgroups are also spread throughout the Microsoft news server like those in the `microsoft.public.inetsdk.programming.components` newsgroup folder.

Here's the simplest definition of ActiveX that you'll ever hear: ActiveX is an advanced form of OLE Control eXtension (OCX). An ActiveX component is 50 to 75 percent smaller than an OCX of the same functionality. In addition, ActiveX components support incremental rendering and asynchronous connection features required for Internet use. Finally, most ActiveX components are faster than their OCX counterparts because an ActiveX component uses a lower-level interface. As the OLE in OCX indicates, an OCX goes through an extra processing layer by using OLE—ActiveX has direct access to COM.

The OCX was originally designed to create components for Visual Basic and represents a very simple implementation of COM technology (maybe a component form of OLE is a better way to look at it). These components were designed exclusively for desktop application use. The OCX, in turn, was originally designed as a 32-bit replacement for the 16-bit Visual Basic eXtension (VBX). Microsoft's original use for the VBX was a technology for creating controls that a programmer could place on a form to create applications with relative ease. These controls represented a new way to write applications—you just built an application's interface using graphics, much as a graphics artist would use primitives to create a drawing.

It's easy to see that ActiveX is yet another example of the evolution of COM as Microsoft learned about programmer's needs. However, this simple definition of ActiveX doesn't even begin to scratch the surface of what you'll actually find under

the hood of an ActiveX control. (We visited ActiveX control creation for the first time in Chapter 10 and will visit it again in Chapter 12.) ActiveX controls include a lot of features earlier component forms didn't. For example, you can use ActiveX to create both windowed and nonwindowed components. In addition, an ActiveX component doesn't necessarily have to have a user interface (as we saw in Chapter 10 when creating a component for server-side scripting).

So, how does ActiveX affect the applications you write? The answer is fairly complex because of the number of ways in which ActiveX gets used, not because the technology itself is so overwhelming. For the most part, ActiveX components are self-contained modules that you can use to create applications. Think of ActiveX in the same way as you would think of building blocks. Both represent basic objects that have no real functionality on their own. For example, a pushbutton component by itself wouldn't make much of an application even though it is a self-contained, fully operational module that's commonly created using ActiveX. However, when you combine a group of ActiveX controls together, then add some glue code to make them do something as the result of user input, you have a functional application in which the individual components work as a single unit.

ActiveX components follow the COM specification fully. When you install an ActiveX control into your programming environment, what have you really accomplished? In most cases, you have a new control that you stick on a form somewhere. You really don't need to know about the control's inner workings. The only important factors are what the control will do for your application and how you interact with it. Again, you're calling a particular module of code installed on your machine using a standard interface—that's what COM is all about.

ActiveX controls are in-process servers. This means that ActiveX relies on the client application for a container in which to store itself and that it has no application-specific capabilities of its own from a traditional perspective. ActiveX controls are essentially DLLs with very special capabilities that are not designed for standalone use. The fact that an ActiveX control is an in-process server means that it relies on the client's address space for resources like memory, and that an errant ActiveX control can crash the entire application.

The building-block approach to programming has other benefits as well. For example, what happens if you have an errant component that doesn't work as anticipated? ActiveX enables you to correct the errant component without recompiling the application. All you need to do is correct the error, recompile the component, place it on the affected machines, then register the new version of the component. That's right, ActiveX controls use the Windows registry, just like every other form of COM. The registry entries identify the ActiveX component, specify its location, and define certain component characteristics like whether the component is scriptable.

There's one other significant difference between ActiveX and older technologies. You can use an ActiveX component in a variety of places that aren't in the usual realm of libraries for new types of applications that programmers of yesterday could only imagine. For example, ActiveX is used on the Internet with client-side scripts to develop Web-based applications. In such cases, ActiveX components are added to the Web page itself using the `<OBJECT>` tag. You can also use ActiveX with server-side scripts as we did in Chapter 10. The interesting thing is that the same ActiveX component could be used in a variety of environments without any recoding.

Many other types of COM component types use ActiveX technology on a standalone, out-of-process server basis like the ActiveDocument (previously known as ActiveX Document) application we'll look at in Chapter 12, which is designed for standalone use. ActiveScripting also falls under the ActiveX umbrella. This is the scripting technology that is used by applications like Internet Explorer. The ActiveScript implementation currently ships with two supported scripting languages: VBScript and JScript.

The Hidden Component

Microsoft uses component technology heavily within its own applications and often makes those components available for developer use. The problem for most developers is figuring out whether a component exists and then finding the documentation required to use it. Of course, there's also the problem of whether the component appears on Microsoft's distribution list (should you decide to market an application based on the component) and whether the component will be supported in the future.

In addition to discrete components, sometimes a technology is embedded as part of a product like Internet Explorer. There could be a wealth of functionality at your fingertips depending on the application, but figuring out where the feature is included, when you can use it, and how to use it can be daunting. The interfaces exposed by the product allow you to create ActiveX objects using the same `CoCreateInstance()` method call that you would use with a discrete component.

To give you an idea of just how severe this problem can get, consider some of the interfaces exposed by Internet Explorer 4. For example, most developers aren't aware that there's an Internet Shortcuts API and that you can use it to manipulate the Internet Explorer Favorites folder for your own purposes. You can use this interface for a variety of needs, including creating a set of links for the company intranet. A network administrator could simply add URLs to a Favorites folder using the standard Internet Explorer interface and be certain that those links would appear on the company intranet for everyone to use. In other words, updating the link list for the intranet would become much easier. This particular interface not only allows you to get the URL, but you can execute a command against the URL and set it as well.

Continued

(continued)

The number of objects you can create using Internet Explorer 4 interfaces alone is pretty astounding. Even smaller parts of the application, like the ratings system, are available for use by other applications. The Internet Ratings API makes things like the Rating Access Denied dialog available for your use. A network administrator could use this interface for all kinds of purposes, including denying employee access to Web sites that the company considers nonjob-related on the Internet. Instead of placing this Web-site list within each employee's browser, you could place it within a SQL database on a main server and provide a utility for changing the database settings. (Interestingly enough, the Rating Access Denied dialog can be set to allow an employee to enter a special user name and password to override the setting when necessary.)

The bottom line for most developers today is that it pays to discover the potential for existing solutions to a problem before you reinvent the wheel. If you see a feature that you find beneficial in a problem, then take the time to find out if there's a documented interface for using that feature. In many cases, finding the hidden component that already exists will save you a lot of time and effort in creating a new component of your own.

DCOM

Microsoft's distributed component object model (DCOM) is the next step in the evolution of COM. It was originally called Network OLE, but like many things COM, Microsoft changed the name. This version of COM will enable components and applications to communicate over a network, no matter how widely separated the participating computers are from each other. DCOM will allow components to work together over a LAN, WAN, or even over the Internet.

DCOM is designed to communicate over a broad range of network protocols. This includes both the HTTP and TCP/IP protocols used by the Internet. In other words, a component on a laptop in New York could communicate with a server running in Paris and use that server's resources as if the two were in the same room. In addition, DCOM takes care of handling the network protocol details for you so that you can concentrate on the business logic of the component that you're designing.

DCOM is modeled after the Open Software Foundation (OSF) distributed computing environment remote procedure call DCE-RPC specification. It is supposed to work with both Java applets and ActiveX controls, but with a COM basis. DCOM is definitely oriented toward ActiveX use.

What DCOM will do for the developer is enable a client application to communicate directly with a service running on a server using standard COM interface. In essence, this means you could design a control or applet that enables the client application to receive data from the server, then perform local calculations using that data. In addition, DCOM allows for secure communication between client and server regardless of the protocol used.



Quite a few resources are at your disposal when it comes to DCOM on the Internet. The best place to start looking for the latest information about this technology is Microsoft's DCOM Web site at <http://www.microsoft.com/com/dcom.asp>. This site provides a link to the DCOM specification, books on how to work with DCOM, programming samples, white papers, and links to other Web sites with additional DCOM information. One of the better places to get information about the OSF DCE-RPC specification (the underlying technology of DCOM) is at http://www.dstc.edu.au/AU/research_news/dce/dce.html. This Web site will help you find the specification, FAQ (frequently asked question) sheets for using DCE-RPC in a variety of environments, and a list of vendors who support DCE-RPC in their products.

Three additions to COM apply specifically to DCOM. The first is the use of a proxy object. The local client uses the proxy object to work with the component. As far as the client is concerned, the component resides on the local hard drive. However, what the client is really working with is a component substitute. The second is a stub that resides on the server. The stub takes the place of the client on the component machine. As far as the component is concerned, the client is local to its machine. The third addition is the communication that goes on between the proxy object and the stub—that's where the DCE-RPC part of the picture comes into play. DCE-RPC enables the proxy object and stub to stay in sync with their real counterparts.

With the mechanisms that DCOM uses to provide service in place, a client never need know whether a component is local or remote. Part of this seamless interface is incorporated into the registry. This form of DCOM access requires you to make new registry entries. The first is an application identifier (AppID) entry in the `HKEY_CLASSES_ROOT\CLSID\<class id>` entry for the requested GUID. This application identifier uses a 128-bit number, just as the GUID does. A `RemoteServerName` key in the application identifier entry tells Windows that it needs to use DCOM to access this component on a remote machine. In sum, even the `CLSID` entry is the same as before—it just points to a remote location instead of a local one.



With Windows 2000, many COM registry entries will reside on the domain controller rather than the local machine when using Active Directory. This will reduce the administrator's workload by providing a central location for registering components and applications. It will allow an administrator to install or move components without reconfiguring every machine on the network. Any change to the COM entries on the domain controller will automatically propagate to all the clients on the network.

In addition to an automated method of creating connections using registry entries, there's a way to specify the server name during object creation for applications that require it using a `COSERVERINFO` data structure. This data structure allows you to specify the server name as a domain naming system (DNS) string, universal naming convention (UNC) string, or TCP/IP address. For example, if you wanted to create an administrative tool that allowed access to multiple servers, it would hardly pay

to specify the remote server name as part of the registry entry. When you need to specify the server as part of the object instance creation process use the `CoCreateInstanceEx()`, `CoGetInstanceFromFile()`, `CoGetInstanceFromStorage()`, or `CoGetClassObject()` functions.

The second method of creating a DCOM connection has the added advantage of allowing you to specify the level and type of security access required using a `COAUTHINFO` data structure. The automated method uses the current user's name and password to log onto the server, which may not be enough to gain the access required to create the object in some cases. The `COAUTHINFO` data structure allows you to do things like impersonate someone else. If you set the `COAUTHINFO` data structure to `NULL`, then Windows will use the current user's security information to gain access to the server as usual.

DCOM is used in a variety of ways in today's computing environment. We'll spend more time looking at DCOM in general in Chapter 13. Chapter 24 will give you a better idea of how you can use DCOM to create remote execution tools, and we'll look at what you'd need to do to create a simple remote administration tool for your server.

COM+

COM+ is yet another extension to the existing COM standard. COM+ is also an extension of several other technologies as well, including both DCOM and ActiveX, making COM+ a truly evolutionary product.

So, why develop yet another standard? One of the problems with existing component technologies is that they are difficult to learn and implement. Most of Microsoft's current component technologies require the skills of an advanced programmer who must understand the intricacies of COM to fully implement it. COM+ will make it easier to develop components in several ways. The following list will describe these advantages.

- | | |
|-------------------------|---|
| Language Neutral | COM+ components are completely language neutral. No longer do you need to worry about the component that you wrote in Visual C++ working in Visual Basic. In the past this was a problem. You might write and test the component using Visual C++ only to find out that Visual Basic was expecting something different. |
| Ease of Usage | COM+ components are completely self-describing. This makes the component accessible to any tool or language that understands COM+. In addition, it eliminates the need to create a type library for the component as is required for COM. |

New Features	This feature allows developers to build advanced controls using special COM+ calls. Some of the more important new features include load balancing, in-memory databases, a publish and subscribe events service, and a queued components service.
Fewer Requirements	You'll create a COM+ component using the features of the language you're familiar with. The COM+ tools and runtime will help you turn the resulting classes into COM+ components and applications. This feature eliminates the need for a class factory in your component (at least in most cases)—all you need is the class required to implement the component.
Ease of Programming	The MTS programming model is a vast improvement over past Microsoft offerings. You'll find that you can develop server-based components faster using COM+.
Database Support	One of the most important COM extensions provided with COM+ is databinding, which allows a component to provide database access. Adding database access means that you can use COM+ to build components for the Internet that can access remote data.
Common Data Types	An ongoing problem with component technology is that different languages use varying data types. In addition, even if two languages do support a common data type, it's very likely that the way the data type is support will be different. COM+ eliminates this problem by supporting one set of common data types that anyone developing components will use. Since the COM+ specification states what these data types look like, they can be duplicated using native-language data types or data-mapping between COM+ types and native-language data types. In addition, COM+ will attempt to coerce any data type that it doesn't recognize into the appropriate COM+ data type. COM+ does allow you to extend the defined data types by defining enums, classes, and structures using native COM+ data types.

COM+ is the basis for Distributed interNet Applications (DNA) architecture, which is Microsoft's latest vision for the Internet. Theoretically, DNA will allow COM components to execute on any platform and will allow the developer to use any language to write the component. As previously stated, however, the language and

tools used by the developer must have an understanding of COM+ in order to make the new ease-of-programming features a reality. In addition, much of the new technology is based on Microsoft Transaction Server and related server technologies.



Since COM+ is such a new technology, your ability to find much information about it, even on the Internet, will be limited. You can, however, go to the main COM+ Web site at <http://www.microsoft.com/com/complus.asp> to find the latest Microsoft-supplied information. This Web site contains pointers to the latest information about COM+, Web sites that support COM+, white papers, articles in the press, and a variety of Microsoft-supplied materials.

Part of the ability to work with any language and any platform will come from a new COM+ feature called *interceptors*. Using interceptors will allow a COM+ component to call a variety of services during runtime rather than relying on a specific service. You'll use interceptors to receive and process events related to instance creation, calls and returns, errors, and instance deletion. Interceptors also provide the mechanism to perform transactions and system monitoring.

Microsoft didn't want to make creating COM+ components that much different from anything programmers have seen in the past. In fact, the goal with COM+ is to keep everything as close to being the same as before and still reduce the complexity of creating the code required for a COM+ component. With this in mind, Visual C++ programmers will be happy to know that you can use COM+ with the relatively minor addition of a new keyword to your vocabulary — *coclass*. The *coclass* keyword is the starting point for creating a COM+ component based on the same skills you use for creating classes with Visual C++ now. (We'll see how to work with COM+ components in Chapter 12.)

COM+ has a few limitations that really make sense when you think about it. The most important limitation is that all properties must use accessor functions (get and read functions). Using accessor functions means that COM+ can always tell when a property has been read or modified by the client. However, you don't have to use properties for every element of your component. Internally you can use fields, which aren't accessible by the client. Fields are maintained by your class for maintaining state information and aren't tracked by COM+ at all.

As you can see, COM+ is a big improvement over previous versions of Microsoft's component technology. Chapters 12 and 13 will show you COM+ in action. If you want to see COM+ on the server, then check out Chapters 15 and 16 where we'll use COM+ with MTS and Microsoft Message Queue (MSMQ) Server.

Working with Interfaces

Working with COM components is all about communication of some type. You want to create some code that is easy to reuse because it can communicate with the client using it and adapt, if necessary. ActiveX-control communication consists of a client, a server, and an interpreter named COMPOBJ.DLL between them. Essentially, these three objects are the basis of what we'll discuss throughout the book because they encompass the three objects that most people work with.

**Tip**

If you're absolutely and totally confused by all of this talk about interfaces, make sure you spend plenty of time reading through the various pieces of OLE documentation that Microsoft provides—the most important of which is the OLE2 SDK. You'll also want to invest in a magazine like *Microsoft Systems Journal*. This particular magazine painstakingly explains most of the new interfaces that Microsoft designs. Finally, make sure you get IDG's latest book on OLE technology, *OLE DB and ODBC Developer's Guide*.

That's not all there is to know about the communication, though—an ActiveX control (a COM component of any kind for that matter) needs to provide a standard interface to make it useful. When an application instantiates a copy of an ActiveX control (the component object), it receives a pointer to a list of functions. That list of functions is housed in what's termed an interface. To make it easier for you to understand what an interface is (at least in the context of an ActiveX control), think of it as a set of semantically related functions implemented as part of a component object. You'll normally see an interface as an array of functions defined in the OLE2 headers for your programming language.

An interface can perform a variety of tasks. For example, you might add a data operation interface like `GetData` or `SetData`. The more complex an ActiveX control, the more interfaces it requires to perform its task.

There's one interface called `IUnknown` that every ActiveX control must provide. It's the basis of all other interfaces—every other interface inherits from this basic interface. This is the interface that gets passed back to your application when you instantiate a copy of the control as a component object. Within this interface (and every other interface supported by an ActiveX control) are the three function calls listed below:

<code>QueryInterface()</code>	This function allows the application to determine what interfaces the object supports. If the application queries an interface and the ActiveX control supports it, the application receives an array of pointers to the functions supported by the interface. Otherwise, the application receives a null pointer.
-------------------------------	--

- AddRef()** This function creates a new relationship to a component object interface. Using this function creates another pointer to the array of function pointers supported by the interface. A component object maintains a reference count of the number of relationships that it has established. The component object gets unloaded from memory only when the reference count is 0.
- Release()** This function allows you to destroy the relationship between an application and an ActiveX control. It decreases the reference count by 1. If the component object's reference count is 0, then this function call also requests the destruction of the component object.

The presence of `IUnknown` means that your application can communicate with any component object it encounters. If your application recognizes the interfaces that the component object supports, it can communicate with that object at a fairly high level. For example, an ActiveX control provides very specific interfaces that your programming environment will know about if it supports them. If your application recognizes only a few of the interfaces, it still might be able to communicate with the component object, but at a much lower level.

The OLE2 specification already supports a lot of interfaces for specific kinds of component objects. The Microsoft OLE2 SDK provides a full synopsis of all of these component object types and the interfaces that they're required to support. We'll also cover a few of the ActiveX-specific requirements throughout this book. However, just because Microsoft hasn't defined a particular interface doesn't mean that you can't define it yourself. The specifications we'll talk about throughout this book, and those that you'll learn from other sources, are the minimum number of interfaces that you can implement. Nothing says that you can't implement more interfaces, then publish an API that tells how to use them. That's the beauty of using COM—you can extend it as needed to meet specific requirements.

Using the OLE/COM Object Viewer Utility

Every OLE object you create, whether it's an application or an ActiveX control, relies on interfaces of some sort. Even language extensions like ActiveX Scripting rely on interfaces. An interface is a method for bundling functions in a way that is independent of programming language. Interfaces are one of the basics of OLE. In addition to the custom interfaces you'll create for your object, every object also supports standard interfaces like `IUnknown`.

Visual Studio provides a handy utility named OLE/COM Object Viewer, which you can use to see these interfaces in more detail. (You might see this utility under a variety of other names—OLE View appears to be the most common alternative name that Microsoft has used.) We'll use this utility several times in the book, so you may want to install it if you haven't done so already.

Note

The latest version of the OLE/COM Object Viewer is 2.10.057, which wasn't shipped as part of Visual Studio 6.0. You can get this update from the main COM Web site at <http://www.microsoft.com/com/default.asp>. This section of the chapter uses the latest version of the OLE/COM Object Viewer, so some screen shots may look slightly different from the one's that you're seeing if you're using an older version of the product. Because the latest version of the OLE/COM Object Viewer fixed several important problems with previous versions of the product, you'll want to download and install the latest version of the product as quickly as possible.

It's important to understand how the OLE/COM Object Viewer can help you during the development process. Say you want to find out about the interfaces provided by a document object server (we talked about one previously in Chapter 3 called MDI.EXE — the source for this example is provided on the CD). Go ahead and open the OLE/COM Object Viewer. You'll see a set of folders that encompass the various types of objects. Open the Document Objects folder, and then open the MDIPE Document folder (or another document objects folder if you don't have the MDI.EXE program installed on your machine). You'll see a list of interfaces that MFC implemented for you as you built the application, as shown in Figure 11-4.

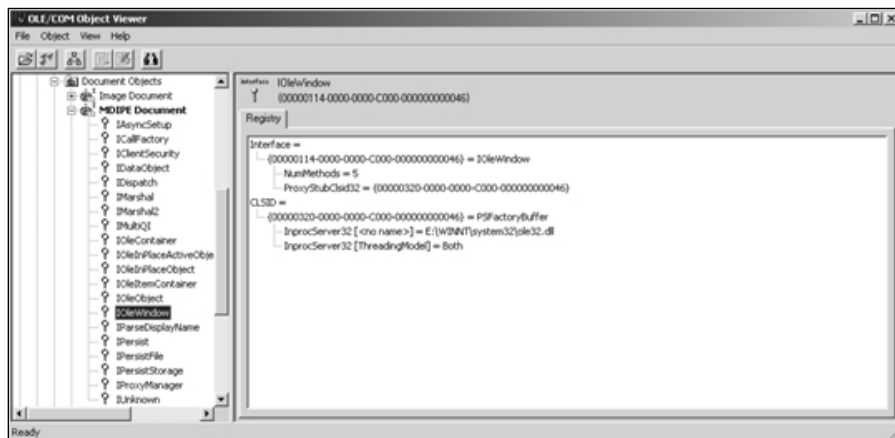


Figure 11-4: The OLE/COM Object Viewer shows a hierarchical view of objects starting with the object type, then the name, and then the interfaces the object supports.

Notice that I've highlighted the `IOleWindow` interface in the left pane. If you look in the right pane, you'll see that this interface has five methods. In addition, you can find the class ID of the proxy stub for this interface. Further research would reveal that one of the methods for the `IOleWindow` interface is `GetWindow()`, which is what a client would call to create a new application window. Because `IOleWindow` is a standard interface, you can be certain that it will always contain the `GetWindow()` function — this is just part of the interface's standard package.

This bit of research would tell you something about this application in addition to the fact that it supports the `IOleWindow` interface. The presence of an `IOleWindow`

interface without the `IActiveScriptSiteWindow` interface means that your application supports OLE automation, not ActiveX Scripting. So, you'd have to write an application that interfaces with this one in such a way that it wouldn't rely on ActiveX Scripting. Let's extend this research further. If you look at every other application in this list, including Office 97 if you have it, you won't find a single application that supports ActiveX Scripting—yet. As you can see, with the proper knowledge of interfaces, you can use OLE/COM Object Viewer to perform all kinds of research on various objects without building a test application first.



Strange things can happen if you create an instance of an application or control in the OLE/COM Object Viewer and then don't release it. For example, your machine might freeze unexpectedly. Every time you view the interfaces supported by an application or control, you have to create an instance to do it. You can tell if there's an instance of an object by looking at the application name. The OLE/COM Object Viewer displays any open objects with bold type. To release the instance of the object you created, right-click the object name (like `MDIPE Document`), and then choose `Release Instance` from the context menu. Fortunately, the OLE/COM Object Viewer is good about closing instances of objects before you leave, but you may need to do this during a viewing session if your machine begins to run out of memory. Remember that every instance you create also uses some memory.

Let's pursue the ActiveX Scripting line of research a bit further. The client-side support for ActiveX Scripting is available. Release the instance of the `MDIPE Document`, and then close the `Document Object` folder. Now open the `ActiveX Scripting Engine` folder. You'll see several entries dedicated to the various scripting languages installed on the host machine. At a minimum you'll see entries for the two default scripting languages: `JavaScript (JScript)` and `VBScript`. All of these entries are for Internet Explorer versions 4.0 and above, but expect to see Netscape products implement them as well. Open the `JScript Language (JavaScript)` entry and you'll see a typical list of interfaces like the one shown in Figure 11-5 (your list may vary slightly from the one shown).

Notice the highlighted entry, `IActiveScript`. This is the first of two client-side interfaces you need to implement client-side ActiveX Scripting. Below it you'll see the `IActiveScriptParse` interface entry, which is the second client-side interface. Having both of these interfaces means that Internet Explorer currently supports ActiveX Scripting as a client.

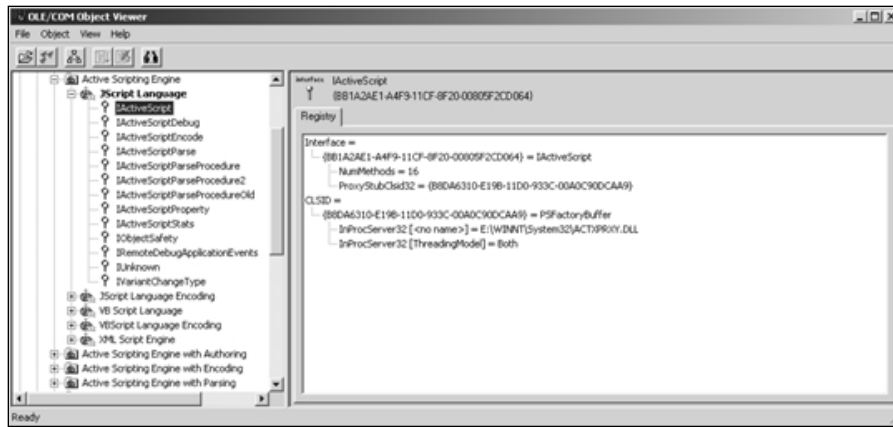


Figure 11-5: ActiveX Scripting client-side support is already available with Internet Explorer.

Summary

This chapter has provided you with a wealth of theoretical information about ActiveX and COM. We've talked about a variety of usage and theoretical concerns that you need to be aware of when using these technologies. In addition, we've looked at one of the most important tools that you'll need to know about when working with both ActiveX and COM. Here are some additional conclusions you can draw from the contents of this chapter.

- ♦ COM offers all of the cumulative effects of the advancements in programming technology like code reuse.
- ♦ In addition to features like code reuse, COM offers many new benefits to the programmer like language independence, extensibility, and platform independence. Many of these features are standard for any object-oriented programming (OOP) methodology.
- ♦ OLE1 and OLE2 aren't different versions of the same product; the underlying technology is totally different. OLE1 relies on DDE, whereas OLE2 relies on COM.
- ♦ OLE2 benefits greatly from using COM as a basis, but the more important features rely on OLE2's ability to view documents as objects rather than imported data from another application.
- ♦ COM is a specification that defines how clients and servers interact.
- ♦ COM objects consist of several components, including methods, properties, events, interfaces, and connection points.

- ♦ Microsoft had several goals when creating COM, including binary compatibility, strong data typing, a base interface, reference counting so that the object could delete itself when no longer needed, a unique identifier, and a special component loader.
- ♦ Always look for the hidden component instead of creating one from scratch. Because most of Windows uses component technology of one type or another, application features are often a clue that a component exists.
- ♦ DCOM allows you to work with components over a variety of network connections using familiar protocols. It relies on OSF DCE-RPC as a method of communication and registry entries to provide a seamless environment for both the client and the component.
- ♦ You can specify the remote server for creating an object when using DCOM in two ways. The first is to include two registry entries that tell Windows which server to use. This method works best when you use the same server to create the object all of the time. The second method is to specify the server during the object creation process using a COSERVERINFO data structure. This method works best for applications like administrative tools where the name of the server will either change or isn't known until the application runs.
- ♦ COM+ is the latest update to Microsoft's component technology. It incorporates a lot of new features, most of which are designed to make COM+ easier to use than its predecessors. In addition, these modifications will have the effect of making COM+ both less prone to bugs and faster.
- ♦ Interfaces are a big part of COM technology of any type. One interface is common to every component—IUnknown. This interface includes three functions that form the basis of working with a component at the lowest level. QueryInterface() allows you to determine which interfaces the component supports. AddRef() is called when you create a new instance of the component, and Release() is called when you destroy an instance of the component.
- ♦ Objects are automatically destroyed when their reference count reaches 0.
- ♦ The OLE/COM Object Viewer utility allows you to make some sense of all those COM registry entries. It allows you to determine which interfaces a component supports, which GUIDs it uses, and many of the implementation details of the component. In addition, OLE/COM Object Viewer organizes the components by type so that you can work with like components without searching through individual component entries.

